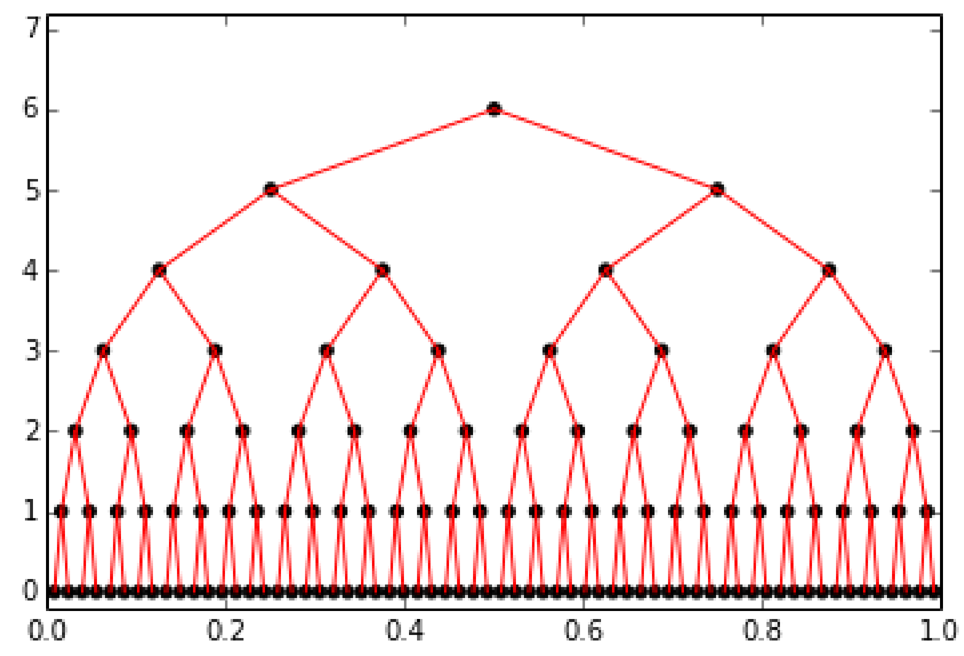
thesis_binary_tree

February 23, 2014

In [1]:
```
from imports import *
import bin_tree_build
np.random.seed(20090403)
```

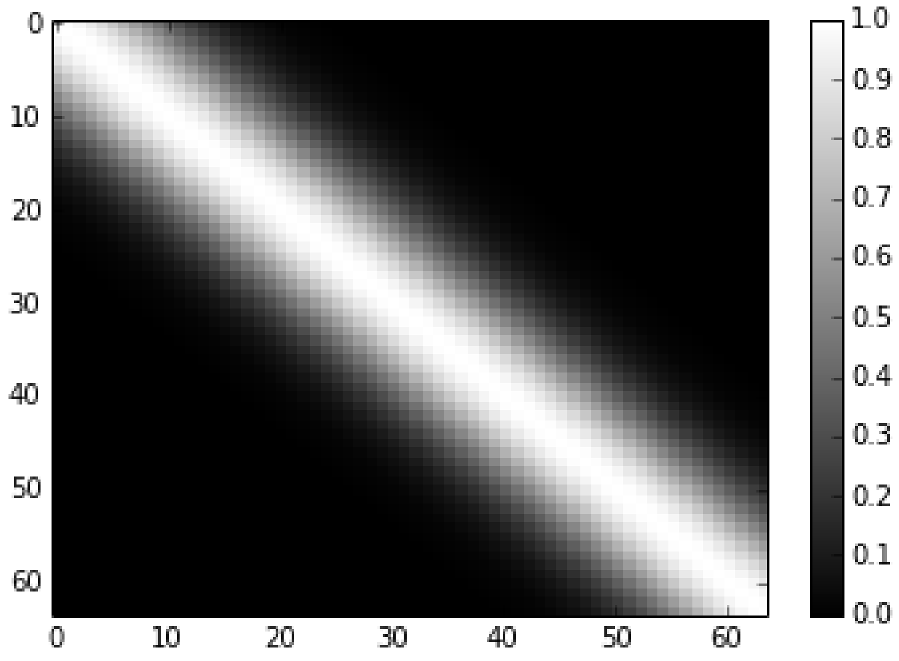
We demonstrate the dyadic binary tree on 64 nodes:

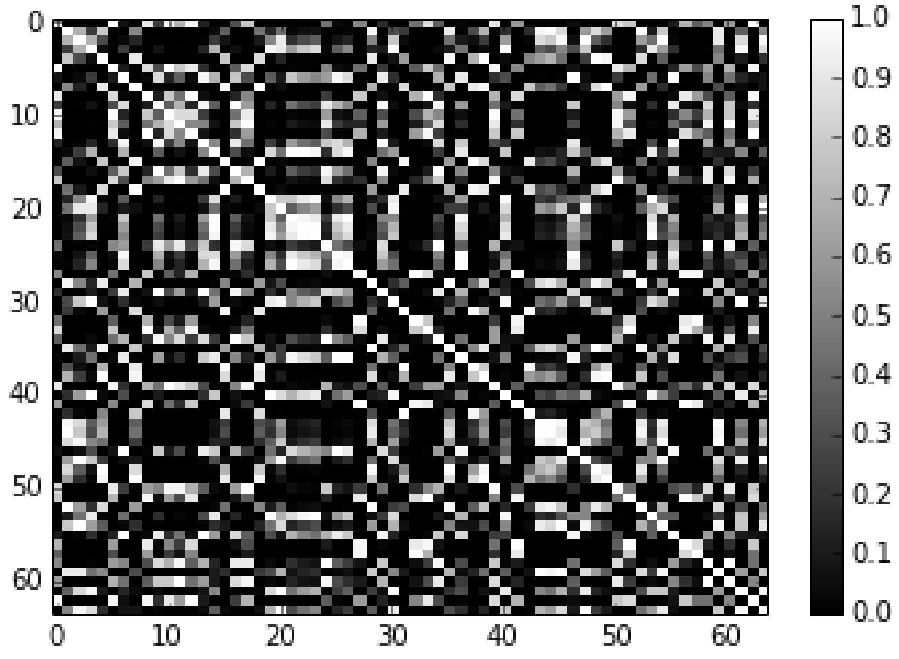
In [2]:
```
dyadic_64 = tree.dyadic_tree(6)
plot_utils.plot_tree(dyadic_64)
```



We generate a toy affinity for demonstration purposes by taking $A(i,j) \sim \exp\left(-\frac{|x_i - x_j|^2}{25}\right)$.
We further consider a random permutation of the points, A_p, and plot the strength of the affinities A and A_p (white is stronger affinity).

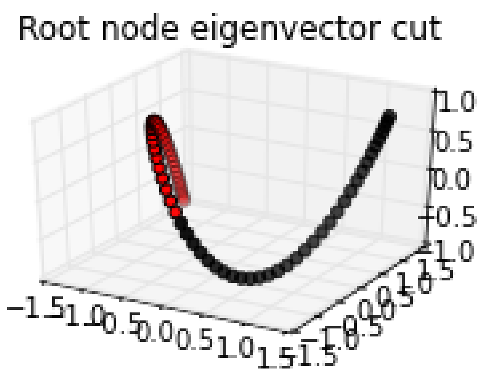
In [3]:
```
A = np.zeros([64,64])
for i in xrange(64):
    for j in xrange(64):
        d_ij = (i-j)
        A[i,j] = np.exp(-(d_ij**2.0/100.0))
row_order = np.random.rand(64).argsort()
A_p = A[row_order,:][:,row_order]
bwplot(A)
plt.colorbar()
plt.show()
bwplot(A_p)
plt.colorbar()
plt.show()
```

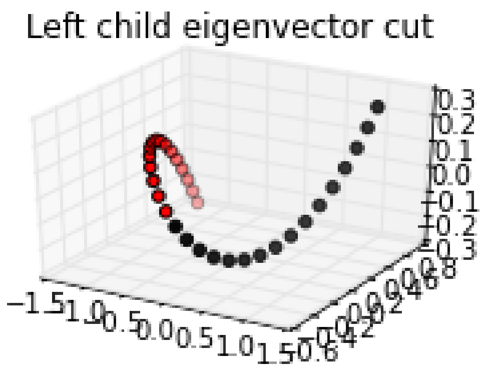


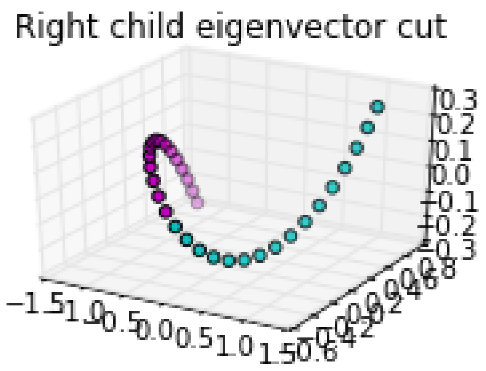


We next construct binary trees on A and A_p. To provide a sense of how the binary cutting process works, we look at the cuts made on the diffusion embedding at diffusion time 1 for the largest few nodes in the tree built on A. The nodes are colored by the split at that level.

In [9]:
```
bt1 = bin_tree_build.bin_tree_build(A,"r_dyadic",1.0)
bt2 = bin_tree_build.bin_tree_build(A_p,"r_dyadic",1.0)

left_node = bt1.children[0]
right_node = bt1.children[1]

fig = plt.figure()
ax1 = fig.add_subplot(221,projection="3d")
ax1.set_title("Root node eigenvector cut")
ax2 = fig.add_subplot(223,projection="3d")
ax2.set_title("Left child eigenvector cut")
ax3 = fig.add_subplot(224,projection="3d")
ax3.set_title("Right child eigenvector cut")

#compute the top few eigenvectors.
vecs,vals = markov.markov_eigs(A,4)
partition = bt1.level_partition(2)
plot_utils.plot_embedding(vecs,vals,diff_time=1.0,partition=partition,ax=ax1)

vecs,vals = markov.markov_eigs(A[left_node.elements,:][:,left_node.elements],4)
partition = np.array(bt1.level_partition(3))[left_node.elements]
plot_utils.plot_embedding(vecs,vals,diff_time=1.0,partition=partition,ax=ax2)

partition = np.array(bt1.level_partition(3))[right_node.elements]
vecs,vals = markov.markov_eigs(A[right_node.elements,:][:,right_node.elements],4)
plot_utils.plot_embedding(vecs,vals,diff_time=1.0,partition=partition,ax=ax3)

fig.tight_layout()

plt.show()
```

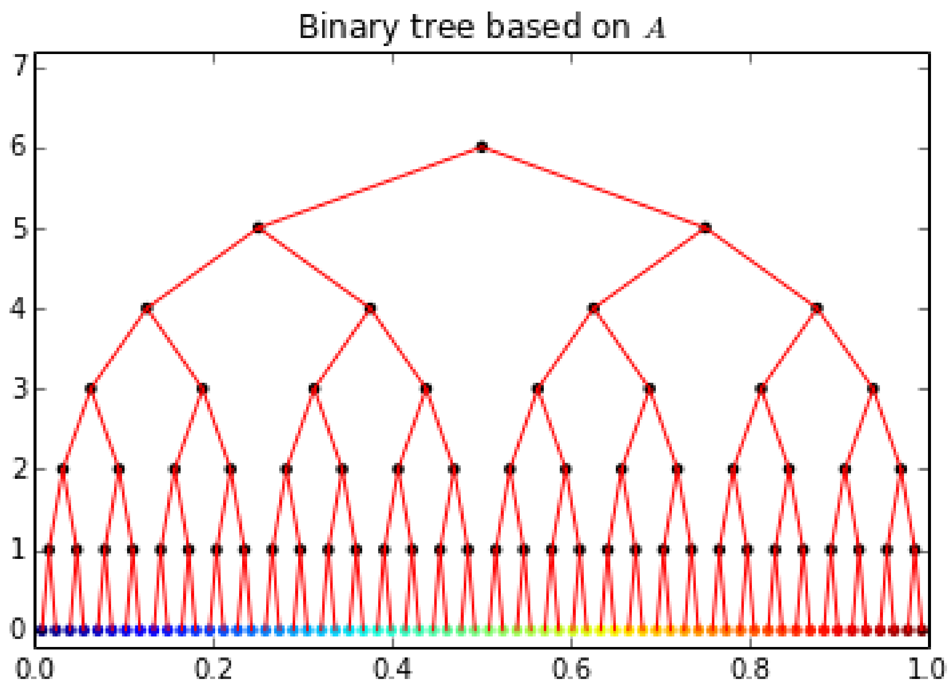


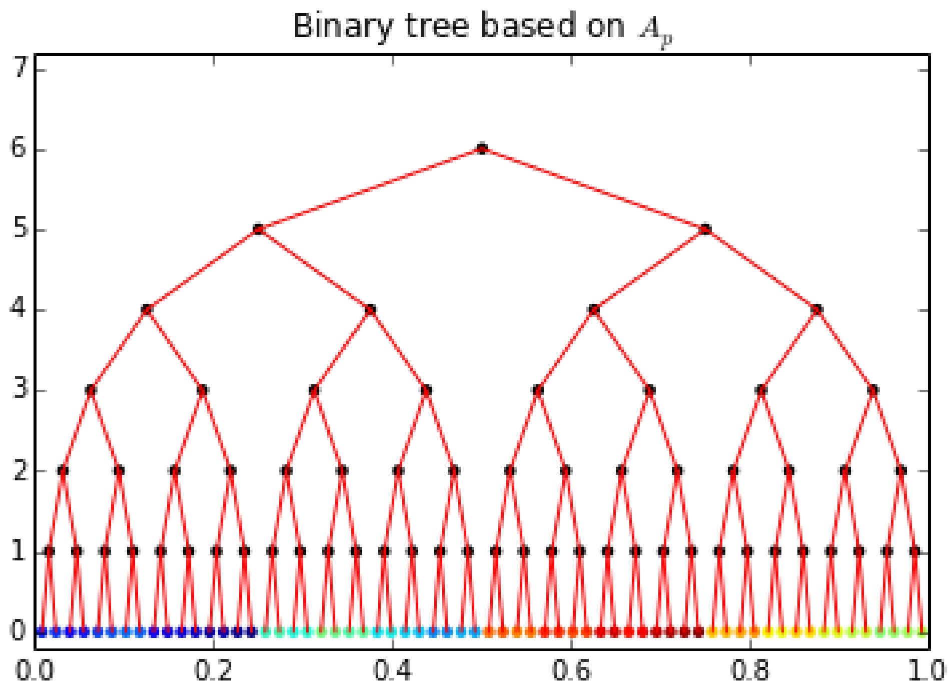




Next we display the trees. We color the leaf nodes on a gradient from 0 to 63 (from A) to show the organization of leaves present in the binary trees.

In [12]:
```
leafcolors = np.arange(-1.0,1.0,2.0/64.0)
plot_utils.plot_tree(bt1,leafcolors=leafcolors)
plt.title("Binary tree based on $A$")
plt.show()

plot_utils.plot_tree(bt2,leafcolors=leafcolors[row_order])
plt.title("Binary tree based on $A_p$")
plt.show()
```





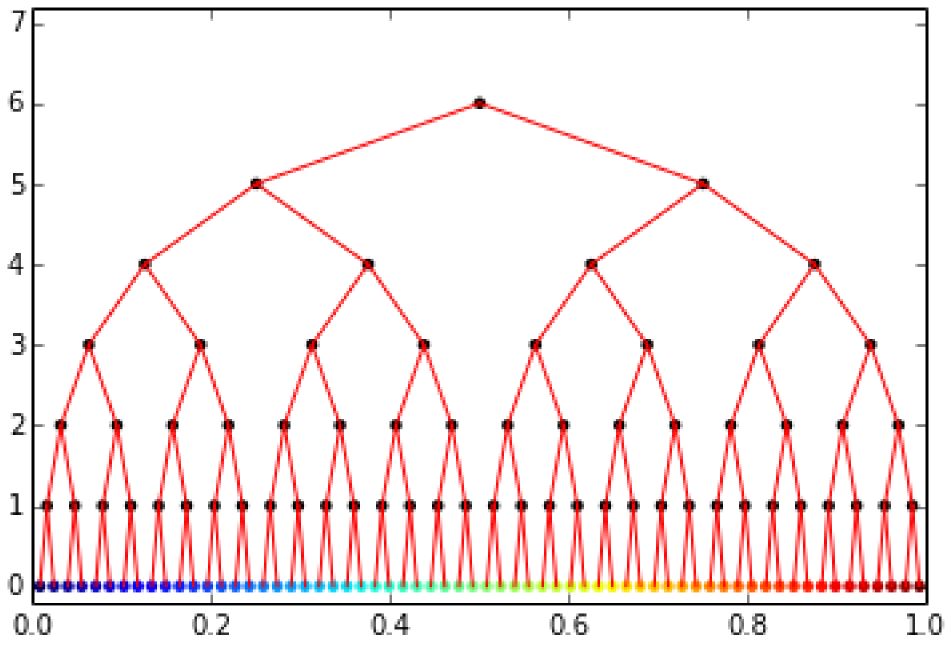
Notice that in the tree built on A_p, the gradient is lost. However, the tree is exactly equivalent, and can be rearranged to match the original tree without any loss of information. In the tree metric, once a split occurs, the internal organization of one of the child nodes is completely irrelevant to nodes outside that child. This can be seen by matching the distance matrix from one tree to the other (after undoing the permutation).

In [16]:
```
inverse_row_order = row_order.argsort()
tree_distances = np.zeros(A.shape)
tree_distances_p = np.zeros(A.shape)
for i in xrange(64):
    for j in xrange(64):
        tree_distances[i,j] = bt1.tree_distance(i,j)
        tree_distances_p[i,j] = bt2.tree_distance(inverse_row_order[i],inverse_row_ord
np.allclose(tree_distances,tree_distances_p)
```

Out [16]:
```
True
```

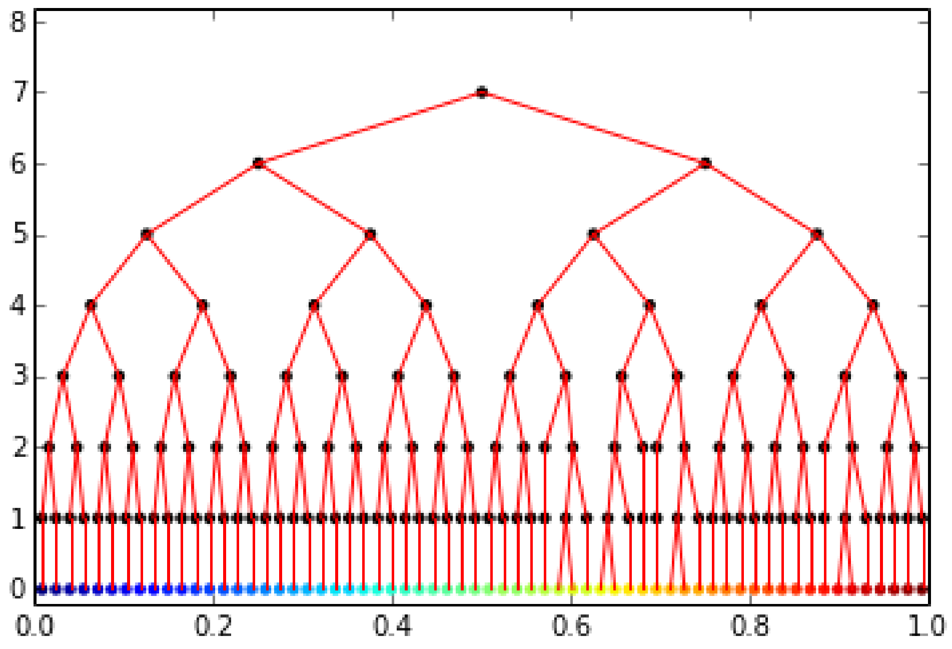
We next demonstrate the other trees discussed in this section. In the dyadic case, the binary splits occur at the median. We can also split the node into two groups at the point where the eigenvector coordinates change sign. Because of the neat symmetry of the original data, splitting A at zero reproduces the original tree exactly.

In [21]:
```
bt3 = bin_tree_build.bin_tree_build(A,"zero")
plot_utils.plot_tree(bt3,leafcolors=leafcolors)
```



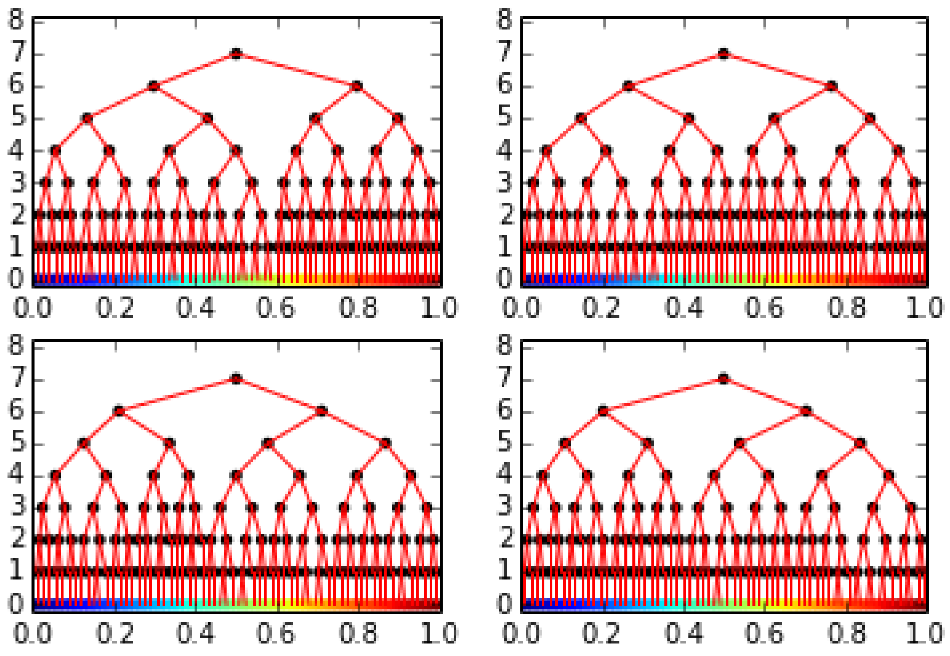
Hence we introduce a slightly noisy version of A by $A_{noise} = (0.9)A + (0.1)U$, where the entries of U are uniform on $[0, 1]$. We then generate the zero-split tree on A_{noise}.

In [22]:
```
A_noise = A*0.90 + np.random.rand(64,64)*0.1
bt3 = bin_tree_build.bin_tree_build(A_noise,"zero")
plot_utils.plot_tree(bt3,leafcolors=leafcolors)
```



This method introduces a perturbation on the trees above that can be desirable for cleaning up artifacts and smoothing boundaries. Finally, we can produce more of this slight randomization by building random dyadic trees. We choose a balance constant (here 1.5), and we can generate many trees which are slight perturbations of each other, but which preserve the overall structure quite well:

In [32]:
```
reload(plot_utils)
fig = plt.Figure()

for i in range(4):
    plt.subplot(2,2,i+1)
    bt4 = bin_tree_build.bin_tree_build(A,"r_dyadic",1.5)
    plot_utils.plot_tree(bt4,leafcolors=leafcolors)
plt.show()
```



```
In []:
```